INFORMATION AND COMPUTATION 75, 39–51 (1987)

Towards Optimal Parallel Bucket Sorting*

TORBEN HAGERUP

FB Informatik, Universität des Saarlandes,
D-6600 Saarbrücken, West Germany

We present a simple deterministic parallel algorithm that runs on a CRCW PRAM and sorts n integers of size polynomial in n in time $O(\log n)$ using $O(n \log\log n/\log n)$ processors. It is closer to optimality than any previously known deterministic algorithm that solves the stated restricted sorting problem in polylog time.

1. INTRODUCTION

It is well known (Ajtai *et al.*, 1983; Cole, 1986; Leighton, 1984) that n objects drawn from an arbitrary totally ordered universe can be sorted by n processors in $O(\log n)$ time, even given a very weak model of parallel computation such as the processor network of bounded degree (assuming, of course, that binary comparisons take unit time). This result is optimal in the sense that the product of the number of processors and the time used is $O(n \log n)$, to be compared with a lower time bound of $\Omega(n \log n)$ for any sequential algorithm operating according to the decision-tree model. Hence no general parallel sorting algorithm that works in $O(\log n)$ time can achieve this with $o(n)$ processors.

The same argument does not apply to *restricted sorting problems*, i.e., if the objects to be sorted are integers drawn from a restricted range, say from the set $\{0, ..., m\}$ with m polynomial in n, such as is almost invariably the case when sorting occurs as part of parallel algorithms for other combinatorial problems. Indeed, if m is polynomial in n, a variant of bucket sort (called radix sort in (Knuth, 1973)) solves the problem in linear sequential time, suggesting that a parallel algorithm sorting in $O(\log n)$ time could conceivably be designed to use only $O(n/\log n)$ processors. While it is not difficult to design such algorithms for $m = (\log n)^{O(1)}$ (Cole and Vishkin, 1986a; Reif, 1985) and, in general, optimal algorithms with a running time of $O(m^\varepsilon + \log n)$ for fixed $\varepsilon > 0$ (Kruskal *et al.*, 1985), the case

* This work was supported by the Deutsche Forschungsgemeinschaft, SFB 124, TP B 2, VLSI Entwurfsmethoden und Parallelität. A preliminary version of this paper was presented at the International Workshop on Parallel Algorithms and Architectures, Suhl, German Democratic Republic, in May 1987.

0890-5401/87 $3.00

of larger values of m simultaneously with a polylogarithmic running time is an unsolved problem of some importance.

Reif (1985) obtained a partial solution by giving a probabilistic algorithm that sorts n integers in the range $\{1, ..., n\}$, uses $O(n/\log n)$ processors, and terminates within $O(\log n)$ steps with high probability. Some doubt remains as to whether his algorithm is able to sort larger numbers, the question hinging on whether the sorting can be made stable. We investigate the restricted sorting problem in a deterministic setting. Although unable to give an optimal algorithm, we do for the first time beat the $\Omega(n)$ processor bound of comparison-based methods by giving an algorithm that sorts n numbers of size polynomial in n in time $O(\log n)$ and uses $O(n \log\log n/\log n)$ processors. This may have implications on other parallel algorithms in which currently sorting is a bottleneck. Our algorithm is to some extent reminiscent of an early algorithm due to Hirschberg (1978).

2. Definitions and Notation

A PRAM (parallel RAM) is a machine consisting of a finite number p of processors (RAMs) operating synchronously on common, shared memory cells numbered 0, 1, ... We assume that the processors are numbered 1, ..., p and that each processor is able to read its own number. All processors execute the same program. We use the unit-cost model in which each memory cell can hold integers of size polynomial in the size of the input and each processor is able to carry out usual arithmetic operations including multiplication and integer division on such numbers in constant time. In addition, we assume the existence of (constant-time) instructions for bitwise logical operations on integers, which are then considered as represented in the binary number system (e.g., 2's complement), as well as for conversion from the unary to the binary number system.

Various types of PRAMs have been defined, differing in the conventions regarding read/write conflicts, i.e., attempts by several processors to access the same memory cell in the same step. *CRCW* (concurrent-read concurrent-write) *PRAM*s allow simultaneous reading as well as simultaneous writing of each cell by arbitrary sets of processors. Simultaneous writing is not immediately logically meaningful, and a further subclassification based on the write conflict resolution rule employed is standard. The variant used in this paper, the *Priority* (*CRCW*) *PRAM*, apparently introduced in (Goldschlager, 1982) and later studied in its relation to other models in (Fich *et al.*, 1985; Kučera, 1982; Vishkin, 1983) and in (Fich *et al.*, 1984) where it is called the MINIMUM model, is the strongest PRAM model commonly considered. It resolves write conflicts by stipulating that in the

event of several processors attempting to write to the same memory cell in the same step, the lowest-numbered processor among them succeeds (i.e., the value that it attempts to write will actually be present in the cell after the write step). Conflicts are thus resolved according to fixed priorities assigned to the processors.

We assume familiarity with basic definitions concerning binary trees. The *depth* of a node u in a tree with root r is its distance from r, i.e., the number of edges on the simple path in the tree from u to r, and the *lowest common ancestor* of two nodes u and v is the common ancestor of u and v of maximum depth.

When a sequence is written using square brackets, as in $x[1], ..., x[n]$, an intended connotation is that the sequence is stored in memory as an array with constant-time access to each of its elements. Finally, the notation $a...b$, for integers a and b with $a \leqslant b$, denotes the set $\{a, a+1, ..., b\}$.

3. The Algorithm

Assume that we are given the input sequence $x[1], ..., x[n]$ of integers in the range $0...m-1$, with $m \leqslant n$. Our task is to compute the permutation π of $\{1, ..., n\}$ such that $x[\pi[1]], ..., x[\pi[n]]$ is sorted in non-decreasing order and such that π has as few inversions as possible (i.e., the sorting is stable). We use the following 3-step procedure whose correctness is obvious:

A. For each $k \in \{0, ..., m-1\}$, compute a list $I[k]$, sorted in increasing order and containing exactly those indices $i \in \{1, ..., n\}$ for which $x[i] = k$.

B. Form the concatenation I of the lists $I[0], ..., I[m-1]$.

C. For each element i in the list I, compute its position $\sigma[i]$ within the list. The position of a list element is one more than the number of elements (properly) preceding it in the list. The desired permutation π is the inverse of σ which can be computed by executing in parallel for all $i \in \{1, ..., n\}$ the instruction $\pi[\sigma[i]] := i$.

The lists $I[k]$ will be represented as explicitly linked structures with constant-time access to the first and last list elements. Hence two lists can be concatenated in constant time. Since list concatenation furthermore is an associative operation, it is easy to carry out step B in $O(\log m)$ time with $O(m/\log m)$ processors and hence in $O(\log n)$ time with $O(n/\log n)$ processors: Each processor begins by sequentially concatenating $\Theta(\log m)$ consecutive lists. The resulting $O(m/\log m)$ lists are combined in the obvious tree-like fashion.

The problem posed by step C is known as the *list ranking problem.* Cole and Vishkin (1986b, 1987) have given two optimal solutions to this problem with running times $O(\log n)$ and processor counts of $O(n/\log n)$. Hence from now on we need only concern ourselves with step A, and step A is the only non-optimal part of the sorting algorithm.

Remark. Since step A needs $\Theta(n \log\log n/\log n)$ processors anyway, one would in practice implement step C via a non-optimal algorithm from (Cole and Vishkin, 1986a) that uses $O(\log n)$ time and $O(n \log\log n/\log n)$ processors, but has the advantage over the two algorithms mentioned above of being very simple.

Note that the computation of a single list $I[k]$ with $O(n/\log n)$ processors is easy; the procedure is much the same as for step B above. The difficulty arises from the fact that we must solve m such problems simultaneously with a total of no more than $O(n \log\log n/\log n)$ processors. Clearly we can devote to the kth problem (the computation of $I[k]$) only a number of processors which is roughly equal to $|I[k]|$ (of course, just computing the numbers $|I[k]|$ is hard; except for the stability requirement, it amounts to solving the sorting problem). We are hence led to consider the following

Neighbour Localization Problem

Input: A set $S \subseteq \{1, ..., n\}$ ($n \geqslant 2$ is called the *size* of the problem) given in the following form: For each $i \in S$, there is a processor P_i with processor number i. The processors P_i, $i \in S$, are the only ones available to solve the problem. Let $S = \{i_1, ..., i_s\}$ with $i_1 < i_2 < \cdots < i_s$.

Output: An array *Next* such that for all j with $1 \leqslant j < s$, $Next[i_j] = i_{j+1}$.

Before we describe a solution to the neighbour localization problem, let us define a slight modification of our model of computation. A *Reversible Priority PRAM* is just like an ordinary Priority PRAM except that the write conflict resolution is programmable in a rudimentary form. In addition to normal write statements of the form "$V := E$" in which the lowest-numbered processor is successful in case of a write conflict, there are reverse priority write statements, symbolically denoted "$V :=^{\text{Rev}} E$", in which the highest-numbered processor succeeds in case of a write conflict.

LEMMA 1. *If the numbers n and $h = \lceil \log n \rceil$ as well as the powers 2^l, $0 \leqslant l \leqslant h$, are precomputed and available to all processors, neighbour localization problems of size n can be solved on a Reversible Priority* PRAM *in time $O(\log\log n)$.*

Proof. The algorithm is best understood by viewing it as operating on a binary tree. Hence let T be the complete, ordered binary tree constructed by the following procedure which also associates an integer number with each node.

(1) Start with a single (root) node numbered 1.

(2) For $i := 2$ to $2^h + n - 1$,

add a new node numbered i as a son of the lowest-numbered node that does not have two sons; if this node is a leaf, add the new node as its left son, otherwise as its right son.

An example is shown in Fig. 1 for $n = 5$.

Let us next introduce some convenient notation related to T. For any subtree T' of T, we use the assertion $u \in T'$ to mean that u is a node in T'. For all $u \in T$, denote the depth of u in T by $depth(u)$ and put $d(u) = h - depth(u)$. For all $l \geqslant 0$ and all $u \in T$ with $depth(u) \geqslant l$, let $p^l(u)$ be the ancestor v of u with $d(v) = d(u) + l$, i.e., the node reached by starting in u and following l father pointers. We abbreviate $p^1(u)$ as $p(u)$. For each non-leaf node $u \in T$, let $T_0(u)$ denote u's *left subtree*, i.e., the maximal subtree of T rooted at u's left son. u's *right subtree* $T_1(u)$ is defined analogously except that it may be empty.

From now on we shall no longer distinguish between a node $u \in T$ and its integer number. For future use we record the following

PROPOSITION 1. (1) *T contains exactly n leaves of depth h. In the order from left to right, they are $2^h, \ldots, 2^h + n - 1$.*

(2) *For all $u \in T$ with $u \neq 1$, $p(u) = \lfloor u/2 \rfloor$. Hence $p^l(u) = \lfloor u/2^l \rfloor$ for all l with $0 \leqslant l \leqslant \text{depth}(u)$.*

(3) *For all $u \in T$ with $u \neq 1$, u is odd if and only if u is the right son of $p(u)$.*

One may associate the numbers $1, \ldots, n$ with the n leaves in T of depth h. This induces a natural association between the available processors and s leaves in T. More precisely, for each $i \in S$ we associate the processor P_i with

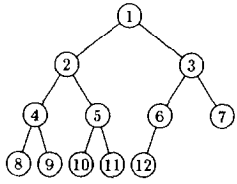

FIG. 1. The tree T for $n = 5$. Each node is labeled by its number.

the leaf $2^h + i - 1$. Whenever convenient in the following, we shall not distinguish between a processor and its associated leaf in T.

The task of each processor P_{i_j} is to "establish contact" with its *neighbours* $P_{i_{j-1}}$ and $P_{i_{j+1}}$. Two neighbouring processors solve this problem by coordinated binary search on their respective sets of ancestors, the goal being to meet at the two processors' lowest common ancestor (this technique was inspired by the $O(\log \log n)$-time priority queue of (van Emde Boas *et al.*, 1977)). To see how this works, assume that there are only two processors L and R, with L to the left of R, and let z^* be their lowest common ancestor.

L and R begin by computing their respective "middle" ancestors $z_L = p^l(L)$ and $z_R = p^l(R)$, where $l = \lfloor (h+1)/2 \rfloor$. L then writes into fields in z_L its processor number together with an indication of the subtree of z_L to which it belongs, and R records its corresponding information in z_R. Now three cases shown in Fig. 2 are possible.

Case (a). $d(z_L) > d(z^*)$; i.e., $z_L = z_R$ is a proper ancestor of z^*. In this case L and R belong to the same subtree of z_L.

Case (b). $d(z_L) = d(z^*)$; i.e., $z_l = z_R = z^*$. Now $L \in T_0(z^*)$ and $R \in T_1(z^*)$.

Case (c). $d(z_L) < d(z^*)$. In this case $z_L \neq z_R$.

Hence both processors are able to determine whether z^* has been found and if not, to correctly continue the search for z^* either above or below the level in T which was "probed." Hence by the properties of ordinary binary search, the lowest common ancestor of L and R will be found within $O(\log h) = O(\log \log n)$ steps. Note how this depends crucially on L and R using exactly the same algorithm for selecting future "probes."

As we shall see, the properties of the Reversible Priority PRAM allow all pairs of neighbouring processors to communicate in the above fashion essentially independently of each other. This is what makes the algorithm work.

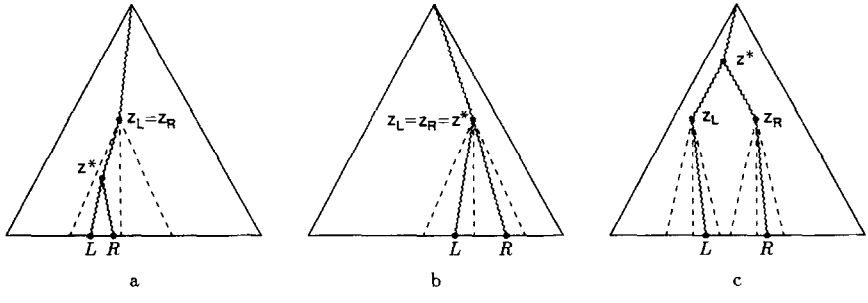

FIG. 2. The three possible relations between z_L, z_R, and z^*.

We now proceed to give a precise description of the algorithm and to prove it correct. This task is greatly simplified if we assume that instead of each processor P_i, we actually have two processors $P_i^{(0)}$ and $P_i^{(1)}$ with the same write priority as P_i. This is permissible since it is obvious that P_i can simulate $P_i^{(0)}$ and $P_i^{(1)}$ at only a constant-factor increase in running time. Informally, $P_i^{(0)}$ will handle P_i's communication to the left, and $P_i^{(1)}$ its communication to the right. Now for each $i \in S$ and each $q \in \{0, 1\}$, the processor $P_i^{(q)}$ executes the program below. Additional explanation is to be found after the program text.

```
(01)  u := 2^h + i - 1;                          (*leaf associated with P_i^(q) *)
(02)  Low := 0;
(03)  High := h + 1;                             (*limits of search interval *)
(04)  for t := 1 to ⌈log(h + 1)⌉                 (*stage count *)
(05)  do begin
(06)       Mid := ⌊(Low + High)/2⌋;
(07)       z := ⌊u/2^Mid⌋;                       (*z = p^Mid(u) = probe node *)
(08)       r := ⌊u/2^(Mid-1)⌋ mod 2;             (*u ∈ T_r(z) *)
(09)       A[z, 1 - r, 0] := NULL;               (*prevent reading garbage in line (15) *)
(10)       A[z, r, 0] := i;                      (*determine leftmost client *)
(11)       A[z, r, 1] :=(Rev) i;                 (*determine rightmost client *)
(12)       if A[z, r, q] ≠ i
(13)       then High := Mid                      (*(a): z is too far from the leaves *)
(14)       else
(15)         if (r = q) or (A[z, 1 - r, 0] = NULL)
(16)         then Low := Mid                     (*(c): z is too close to the leaves *)
(17)         else                                (*(b): z = lowest common ancestor *)
(18)           if q = 1
(19)           then Next[i] := A[z, 1, 0];
(20)  end;
```

Algorithm *Locate-Neighbours.*

A is a 3-dimensional array whose first index ranges over $1 \ldots 2^h - 1$ and whose second and third indices both range over $0 \ldots 1$. A represents the internal nodes of T, with $A[z, \cdot, \cdot]$ representing the node z. Hence each internal node z has four fields $A[z, 0, 0]$, ..., $A[z, 1, 1]$. The variables *Low*, *High*, *Mid*, r, u, and z are supposed to be local to each processor. They are actually represented as arrays: Let X stand for any one of the above six names of local variables. Then a reference to X in the program text actually is a reference to an array element $X[i, q]$; in the interest of readability, this indexing was suppressed in the above description.

Lines (10) and (11) must be executed simultaneously by all processors. The necessary synchronization mechanism was omitted, again in order to not clutter the picture by irrelevant detail. Finally, "NULL" denotes any value (such as 0) that cannot be a processor number.

Let us call an execution of lines (06)–(19) a *stage*. A stage represents one test in each binary search carried out by neighbouring processors. Now for $1 \leqslant j < s$, let z_j^* be the lowest common ancestor of the neighbouring processors $P_{i_j}^{(1)}$ and $P_{i_{j+1}}^{(0)}$, which will be called *mates*, and for $t = 0, \ldots, \lceil \log(h+1) \rceil$, let $Q(t)$ and $W(t)$ denote the assertions

$Q(t)$: At the end of the tth stage, the following relations hold for all j, $1 \leqslant j < s$: $Low[i_j, 1] = Low[i_{j+1}, 0]$, $High[i_j, 1] = High[i_{j+1}, 0]$, and at least one of the conditions (α) and (β) below is satisfied.

(α) $0 \leqslant Low[i_j, 1] < d(z_j^*) < High[i_j, 1] \leqslant h+1$ and $High[i_j, 1] - Low[i_j, 1] \leqslant 2^{\lceil \log(h+1) \rceil - t}$.

(β) $\left\lfloor \dfrac{Low[i_j, 1] + High[i_j, 1]}{2} \right\rfloor = d(z_j^*)$ and $Next[i_j] = i_{j+1}$.

$W(t)$: $1 \leqslant \left\lfloor \dfrac{Low[i_1, 0] + High[i_1, 0]}{2} \right\rfloor \leqslant h$ and

$1 \leqslant \left\lfloor \dfrac{Low[i_s, 1] + High[i_s, 1]}{2} \right\rfloor \leqslant h.$

In view of the slightly forbidding notation, a word of explanation is in order. The first part of Q states that each pair of mates is properly coordinated. Condition (α) says that the invariant for binary search is not violated and that the search is making progress, and condition (β) says that the lowest common ancestor has been found. W simply expresses that the extreme processors $P_{i_1}^{(0)}$ and $P_{i_s}^{(1)}$ which have no mates do not get in the way of the other processors. Both $Q(0)$ and $W(0)$ should be interpreted as assertions about values of the program's variables at the beginning of the first stage.

CLAIM. $Q(t)$ and $W(t)$ are true for $t = 0, \ldots, \lceil \log(h+1) \rceil$.

Proof. By induction. $Q(0) \wedge W(0)$ is trivially verified. Now assume $Q(t-1) \wedge W(t-1)$ for some t with $1 \leqslant t \leqslant \lceil \log(h+1) \rceil$. We will show the truth of $Q(t)$ and leave the proof of $W(t)$ to the reader.

Let us take a closer look at what happens in the tth stage. Thus by convention, each program variable name used below in an arithmetic expression denotes the value of that program variable at the end of the tth stage, and each assertion about the execution of particular program statements refers to that execution of the statements which takes place during the tth stage.

Consider briefly a fixed processor $P_i^{(q)}$ and interpret X as $X[i, q]$ for each local variable name X. By the inductive hypothesis, $1 \leqslant Mid \leqslant h$. Hence the relations below, which have also been recorded as comments in the program text, follow easily from Proposition 1.

(i) $z = p^{Mid}(u)$.

(ii) $u \in T_r(z)$.

We may proceed, using (i) and (ii) as well as the definition of the Reversible Priority PRAM model, to give the following interpretation of program lines (10)–(11):

(iii) Each internal node $z \in T$ has a set of *clients*, namely those descendant processors $P_i^{(q)}$ for which $z[i,q] = z$. For $r = 0, 1$, the processor number of the leftmost (rightmost) client in $T_r(z)$ is recorded in $A[z, r, 0]$($A[z, r, 1]$, respectively) during the execution of lines (10)–(11).

The situation described in (iii) is depicted in Fig. 3.

Since $Q(t)$ is the conjunction of claims $Q(t, j)$ for $1 \leqslant j < s$ (with $Q(t, j)$ defined in the obvious way), let us prove $Q(t, j)$ for one arbitrary but fixed j, $1 \leqslant j < s$. Thus we are essentially dealing with two processors $L = P_{i_j}^{(1)}$ and $R = P_{i_{j+1}}^{(0)}$, and we may simplify the notation by replacing $X[i_j, 1]$ and $X[i_{j+1}, 0]$ by X_L and X_R, respectively, for each local variable name X. If $X_L = X_R$, we shall feel free to omit the subscript altogether. Finally, let $z^* = z_j^*$. The phrase "the test in line *ln* succeeds (fails) in P", whereby *ln* is a line number and P a processor, will be used as shorthand for "in the program execution carried out by P, the test in line *ln*, when performed in the tth stage, yields the result **true**(**false**)".

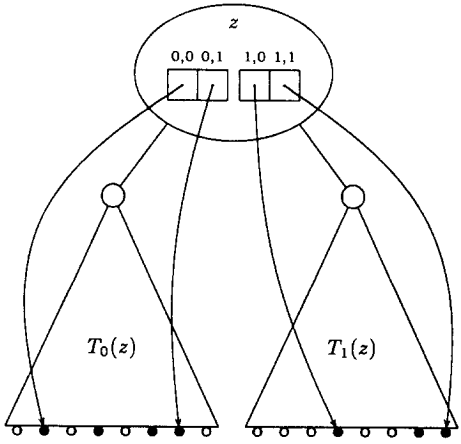

FIG. 3. The four fields of a node z and their use. Clients of z are shown as black nodes.

Observe that by the inductive hypothesis, $Mid_L = Mid_R (= Mid)$. We consider again the cases shown in Fig. 2.

Case (a). $Mid > d(z^*)$. Then by (i) and (ii), $z_L = z_R (= z)$ and $r_L = r_R$. Hence L and R are both clients of z and belong to the same subtree T' of z. But then L is not the rightmost and R is not the leftmost client of z in T', causing the test in line (12) to succeed in both L and R and both processors to execute the assignment in line (13) (i.e., L and R agree to search for z^* closer to the leaves). This is easily seen to imply $Q(t, j)$, with condition (α) true.

Case (b). $Mid = d(z^*)$. Now $z_L = z_R = z^* (= z)$, $r_L = 0$ and $r_R = 1$. Since L is the rightmost client of z (indeed, the rightmost processor) in $T_0(z)$ and R is the leftmost client of z in $T_1(z)$, $A[z, 0, 1] = i_j$ and $A[z, 1, 0] = i_{j+1}$ after the execution of line (11). But then the tests in lines (12) and (15) fail in both L and R (i.e., L and R agree that z^* has been found), and in line (19) the value $A[z, 1, 0] = i_{j+1}$ is assigned to $Next[i_j]$ by L. Hence $Q(t, j)$ holds with condition (β) true.

Case (c). $Mid < d(z^*)$. As in case (b), the test in line (12) fails in both L and R. If $r_L = 0$ (i.e., $L \in T_0(z_L)$), then $T_1(z_L)$ contains no processors, in particular, no clients of z_L. Hence no writing to $A[z_L, 1, 0]$ can take place in line (10), which means that the value NULL stored there by L in line (09) will survive until the test in line (15). Thus whatever the value of r_L, the test in line (15) succeeds in L, causing it to execute line (16). By symmetry, the same holds for R (i.e., L and R agree to search for z^* closer to the root). As in case (a), one easily checks that $Q(t, j)$ is satisfied with condition (α) true.

This ends the proof of the claim.

Now consider the true statement $Q(\lceil \log(h+1) \rceil)$. Its condition ($\alpha$) cannot be true for any j, $1 \leqslant j < s$, since $Low[i_j, 1] < d(z_j^*) < High[i_j, 1]$ implies $High[i_j, 1] - Low[i_j, 1] \geqslant 2$, whereas $High[i_j, 1] - Low[i_j, 1] \leqslant 2^{\lceil \log(h+1) \rceil - t}$ with $t = \lceil \log(h+1) \rceil$ implies $High[i_j, 1] - Low[i_j, 1] \leqslant 1$. Thus for all j with $1 \leqslant j < s$, $Next[i_j] = i_{j+1}$ after the execution of algorithm *Locate-Neighbours*. Since the execution time is clearly $O(\log\log n)$, we have proved Lemma 1. ∎

LEMMA 2. *For $m \leqslant n$, n integers in the range $0 \ldots m-1$ can be stably sorted in $O(\log n)$ time by a Priority PRAM using $O(n \log\log n / \log n)$ processors and $O(mn)$ space.*

Proof. We need only describe the implementation of step A. In essence, the method is to solve m neighbour localization problems, one for each possible input value. Assume briefly that there are n processors $P_1, \ldots, P_n$.

The array A used by algorithm *Locate-Neighbours* is extended by a fourth index (in the first position, say) ranging over $0...m-1$, and each reference to A in the program of some processor P_i is replaced by a reference to the "slice" $A[x[i], \cdot, \cdot, \cdot]$ obtained by fixing the first index of A at the value $x[i]$. This concludes the proof of Lemma 2, except that we still have to pay attention to the following complications:

(1) Although algorithm *Locate-Neighbours* comes close to computing the desired lists $I[k]$, we still have to establish pointers to the first and last elements of the list (in the context of the algorithm, we must compute i_1 and i_s).

(2) The solution presented to the m neighbour localization problems so far uses n processors, whereas only $O(n \log\log n/\log n)$ processors are available.

(3) Lemma 1 assumes certain quantities to be precomputed.

(4) It also uses the stronger Reversible Priority PRAM model.

Let us deal with the problems in order:

(1) i_1 and i_s are easily computed in two steps by the Reversible Priority PRAM.

(2) The resource bounds of $O(\log\log n)$ time and $O(n)$ processors may be trivially changed through processor multiplexing to yield $O(\log n)$ time and $O(n \log\log n/\log n)$ processors as claimed in the lemma.

(3) The assumed precomputation presents no problem since the values in question are the same for all processors and can be computed sequentially in $O(\log n)$ steps.

(4) A p-processor Reversible Priority PRAM is easily simulated by an ordinary p-processor Priority PRAM with only a constant-factor loss in speed. For $i = 1, ..., p$, let P_i be the processor numbered i. A reverse priority write step is implemented simply by letting P_i, for $i = 1, ..., p$, execute the write instruction of P_{p+1-i}. Addresses of cells to be updated as well as values to be written may be exchanged between P_i and P_{p+1-i} in a preliminary step. ▮

Remark. The assumption of unit-time multiplication and division may be dropped as long as each processor is able to carry out arbitrary bit shifts (i.e., multiplications and divisions by powers of 2) in constant time. The only points where this is not obvious are access to multidimensional arrays and processor multiplexing (in distributing n tasks numbered 1, ..., n among $p \leqslant n$ processors, one wants the ith processor, for $i = 1, ..., p$, to take care of the tasks numbered $\lfloor (n-i)/p \rfloor p + i, (\lfloor (n-i)/p \rfloor - 1)\, p + i, ..., p + i, i$. Hence the quantities $\lfloor (n-i)/p \rfloor p$ must be computed). But in all cases,

it is possible to replace each divisor and at least one factor of each multiplication by a power of 2 of the same magnitude.

Recall that any procedure that stably sorts integers in the range $0 \ldots m-1$ may be applied successively c times, for any $c \geqslant 1$, to sort integers in the range $0 \ldots m^c - 1$. Specifically, express the numbers to be sorted in the positional system with base m and stably sort them first according to their last (least significant), then according to their second-to-last, ..., and finally according to their cth last digit (add leading zeros as necessary). When this has been carried out, the numbers will be (stably) sorted. Hence by letting $m = n^\varepsilon$ in Lemma 2, we obtain:

MAIN THEOREM. *For any fixed $\varepsilon > 0$, n integers of size polynomial in n can be sorted in $O(\log n)$ time by a Priority PRAM using $O(n \log\log n / \log n)$ processors and $O(n^{1+\varepsilon})$ space.*

4. CONCLUSION

The sorting algorithm presented in this paper has some desirable and some less desirable characteristics that have not yet been touched upon. On the positive side, the algorithm is simple and easy to program. It has no large "hidden factors" and is fast in practical terms. Finally, it is uniform and robust: The same program works for all n, the value of m need not be explicitly communicated to the algorithm, and if the input numbers happen to be too large, they are still sorted correctly, although not in $O(\log n)$ time. The main weak points of the algorithm are its superlinear space requirements and the fact that it depends on a very strong model of parallel computation.

The construction of an optimal algorithm for restricted sorting remains an open problem, leaving ample room for possible improvement.

RECEIVED January 1987

REFERENCES

AJTAI, M., KOMLÓS, J., AND SZEMERÉDI, E. (1983), An $O(n \log n)$ sorting network, *in* "Proceedings, 15th Annual ACM Symposium on Theory of Computing," pp. 1–9.

COLE, R. (1986), Parallel merge sort, *in* "Proceedings, 27th Annual Symposium on Foundations of Computer Science," pp. 511–516.

COLE, R., AND VISHKIN, U. (1986a), Deterministic coin tossing with applications to optimal parallel list ranking, *Inform. and Control* **70**, 32–53.

COLE, R., AND VISHKIN, U. (1986b), Approximate and exact parallel scheduling with applications to list, tree and graph problems, *in* "Proceedings, 27th Annual Symposium on Foundations of Computer Science," pp. 478–491.

COLE, R., AND VISHKIN, U. (1987), Faster optimal parallel prefix sums and list ranking, to appear in *Inform. and Computation.*

FICH, F. E., RAGDE, P. L., AND WIGDERSON, A. (1984), Relations between concurrent-write models of parallel computation, *in* "Proceedings, 3rd Annual ACM Symposium on Principles of Distributed Computing," pp. 179–189.

FICH, F. E., MEYER AUF DER HEIDE, F., RAGDE, P., AND WIGDERSON, A. (1985), One, two, three ... infinity: Lower bounds for parallel computation, *in* "Proceedings, 17th Annual ACM Symposium on Theory of Computing," pp. 48–58.

GOLDSCHLAGER, L. M. (1982), A universal interconnection pattern for parallel computers, *J. Assoc. Comput. Mach.* **29**, 1073–1086.

HIRSCHBERG, D. S. (1978), Fast parallel sorting algorithms, *Comm. ACM* **21**, 657–661.

KNUTH, D. E. (1973), "The Art of Computer Programming," Vol. 3, "Sorting and Searching," Addison–Wesley, Reading, MA.

KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. (1985), The power of parallel prefix, *IEEE Trans. Comput.* **C-34**, 965–968.

KUČERA, L. (1982), Parallel computation and conflicts in memory access, *Inform. Process. Lett.* **14**, 93–96.

LEIGHTON, T. (1984), Tight bounds on the complexity of parallel sorting, *in* "Proceedings, 16th Annual ACM Symposium on Theory of Computing," pp. 71–80.

REIF, J. H. (1985), An optimal parallel algorithm for integer sorting, *in* "Proceedings, 26th Annual Symposium on Foundations of Computer Science," pp. 496–504.

VAN EMDE BOAS, P., KAAS, R., AND ZIJLSTRA, E. (1977), Design and implementation of an efficient priority queue, *Math. Systems Theory* **10**, 99–127.

VISHKIN, U. (1983), Implementation of simultaneous memory address access in models that forbid it, *J. Algorithms* **4**, 45–50.